Table of Contents

Starting out

Observables

Operators

Recipes

Advanced

Appendices

Rxjs ultimate

This book is meant to be starter as well as intermediate book for anyone starting out with Rxjs.

Tweet me @chris_noring for questions, cheers, comments. See you out there.

The different articles can be read in any order but it is recommended when you are new to Rxjs to look at the basics like Observable Observer and Producer

This is a work in progress so chapters are likely to be added and changed, less and less so as time passes, but the work was started on this on the 21 feb 2017..

Updated to RxJS 6

This book was updated to RxJS on 28th April 2018.

A lot of things changed when going from RxJS 5 to RxJS 6. Most of the changes was about moving to pipeable operators, A way for the tree shaking process to work better but also do introduce some order. For more detail on the exact nature of the changes https://github.com/ReactiveX/rxjs/blob/master/MIGRATION.md

Give credit where credit is due

If you like using Rxjs /Rxjs 5 be sure to tweet @BenLesh or @mattpodwysocki etc and say thank you, or good job. It means a lot..

I have now synced this book with a GitHub repo. If you wish to contribute or correct something, please raise a PR at https://github.com/softchris/Rxjs5-Ultimate

This book will always be free.

Other titles by me

If you are interested in other things I've authored here is a list of books:

React (free)

Just started with this one in April 2018, a lot more to come.

Another free gitbook where I intend to cover as much about React as possible. Happy to have your help here. Power to the community

-https://legacy.gitbook.com/book/chrisnoring/react/details

Learning Angular

Beginner to Master book on Angular up to version 5

- https://www.amazon.com/dp/1787124924
- https://www.packtpub.com/web-development/learning-angular-second-edition

Architecting Angular Applications with Redux, RxJS, and NgRx

A book covering, Flux, Redux, RXJS and NGRX with Angular

- https://www.packtpub.com/web-development/architecting-angular-applications-redux
- https://www.amazon.com/Architecting-Angular-Applications-Flux-Redux/

Chris Noring

Async code

Async code is code that isn't done immediately when being called.

```
setTimeout(() => {
  console.log('do stuff');
}, 3000 )
```

3s seconds in the future the timeout is done and `do stuff` is echoed to the screen. We can see that the anonymous function we provide is being triggered when time has passed. Now for another more revealing example:

```
doWork( () => {
  console.log('call me when done');
})
```

```
function doWork(cb){
   setTimeout( () => {
     cb();
   }, 3000)
}
```

Other example of callback code are events here demonstrated by a `jQuery` example

```
input.on('click', () => {

})
```

The gist of callbacks and async in general is that one or more methods are invoked sometime in the future, unknown when.

The Problem

So we established a callback can be a timer, ajax code or even an event but what is the problem with all that?

One word **Readability**

Imagine doing the following code

```
syncCode()  // emit 1
syncCode2()  // emit 2
asyncCode()  // emit 3
syncCode4()  // emit 4
```

The output could very well be

```
1,2,4,3
```

Because the async method may take a long time to finish. There is really no way of knowing by looking at it when something finish. The problem is if we care about order so that we get 1,2,3,4

We might resort to a callback making it look like

```
syncCode()
syncCode()2
asyncCode(()= > {
   syncCode4()
})
```

At this point it is readable, somewhat but imagine we have only async code then it might look like:

```
asyncCode(() => {
   asyncCode2(() => {
     asyncCode3() => {

     }
   })
})
```

Also known as `callback hell`, pause for effect :)

For that reason promises started to exist so we got code looking like

```
  getData()
    .then(getMoreData)
    .then(getEvenMoreData)
```

This is great for `Request/Response` patterns but for more advanced async scenarios I dare say only Rxjs fits the bill.

Why Rxjs

Promises lack the ability to generate more than one value, ability to retry and it doesn't really play well with other async concepts.

Installation and Setup

The content for this chapter is taken from the offical docs but I am trying to provide you with a little more information as to why and also its nice to have everything in one place. Official docs

TH Rxjs lib can be consumed in many different ways, namely `ES6` , `CommonJS` and as `ES5/CDN` .

ES6

Install

```
npm install rxjs
```

Setup

```
import { of } from 'rxjs';

of(1,2,3)
```

GOTCHA

This statement `import Rx from 'rxjs/Rx'` utilizes the entire library. It is great for testing out various features but once hitting production this is *a bad idea* as Rxjs is quite a heavy library. In a more realistic scenario you would want to use the alternate approach below that only imports the operators that you actually use :

```
import { of } from 'rxjs';
import { map } from "rxjs/operators";

let stream$ = of(1,2,3).pipe(
  map(x => x + '!!!')
)

stream$.subscribe((val) => {
  console.log(val) // 1!!! 2!!! 3!!!
})
```

CommonJS

Same install as with ES6

Install

```
npm install rxjs
```

Setup

Setup is different though

Below is yet again showcasing the greedy import that is great for testing but bad for production

```
var { of } = require('rxjs');

of(1,2,3); // etc
```

And the better approach here being:

```
var { of } = require('rxjs');
const { map } = require('rxjs/operators');

of(1,2,3).pipe(
  map((x) => { return x + '!!!'; })
) // etc
```

As you can see we go about it a little differently when we grab the `of` operator. It looks pretty much the same when we drill down to get the `map` operator.

Notice that we will most likely find the operator we need from `rxjs/operators` and that we no longer use an `Observable` object

CDN or ES5

If I am on neither ES6 or CommonJS there is another approach namely:

```
<script src="https://unpkg.com/rxjs/bundles/Rx.min.js"></script>
```

Notice however this will give you the full lib. As you are requiring it externally it won't affect your bundle size though.

Observable Anatomy

An observable's subscribe method has the following signature

```
stream.subscribe(fnValue, fnError, fnComplete)
```

The first one is being demonstrated below **fnValue**

```
import { Observable } from 'rxjs';

let stream$ = Observable.create((observer) => {
  observer.next(1)
});

stream$.subscribe((data) => {
  console.log('Data', data);
})

// 1
```

When `observer.next(<value>)` is being called the `fnValue` is being invoked.

The second callback **fnError** is the error callback and is being invoked by the following code, i.e `observer.error(<message>)`

```
let stream$ = Rx.Observable.create((observer) => {
   observer.error('error message');
})

stream$.subscribe(
   (data) => console.log('Data', data),
   (error) => console.log('Error', error))
```

Lastly we have the **fnComplete** and it should be invoked when a stream is done and has no more values to emit. It is triggered by a call to `observer.complete()` like so:

```
let stream$ = Rx.Observable.create((observer) => {
   // x calls to observer.next(<value>)
   observer.complete();
})
```

Unsubscribe

So far we have been creating an irresponsible Observable, irresponsible in the sense that it doesn't clean up after itself. So let's look at how to do that:

```
let stream$ = Observable.create((observer) => {
  let i = 0;
  let id = setInterval(() => {
    observer.next(i++);
  },1000)

  return function(){
    clearInterval( id );
  }
})

let subscription = stream$.subscribe((value) => {
  console.log('Value', value)
});

setTimeout(() => {
  subscription.unsubscribe() // here we invoke the cleanup function

}, 3000)
```

So ensure that you

- Define a function that cleans up
- Implicitely call that function by calling `subscription.unsubscribe()`

Observer

Note the following code example creating an `Observable`

```
import { Observable } from 'rxjs';

let stream$ = Observable.create((observer) => {
  observer.next(4);
})
```

`create` method takes a function with an input parameter `observer`.

An Observer is just an object with the following methods `next` `error` `complete`

```
observer.next(1);
observer.error('some error')
observer.complete();
```

Producer

A producer have the task of producing the values emitted by an Observable

```
class Producer {
   constructor(){
     this.i = 0;
   }

   nextValue(){
     return this.i++;
   }
}
```

And to use it

```
 let stream$ = Observable.create( (observer) => {
   const producer = new Producer();
   observer.next( producer.nextValue() )
   observer.next( producer.nextValue() )
 })
```

In the Observable Anatomy chapter there is no `Producer` in the examples but most `Observables` that are created by a helper method will have an internal `Producer` producing values that the observer emits using the `observer.next` method

Observable vs Promise

Let's dive right in. We have created something called an Observable. An async construct, much like a promise that we can listen to once the data arrives.

```
import { from } from 'rxjs';

let stream$ = from([1,2,3])

stream$.subscribe( (value) => {
   console.log('Value',value);
})

// 1,2,3
```

The corresponding way of doing this if dealing with promises would be to write

```
let promise = new Promise((resolve, reject) => {
   setTimeout(()=> {
      resolve( [1,2,3] )
   })

})

promise.then((value) => {
  console.log('Value',data)
})
```

Promises lack the ability to generate more than one value, ability to retry and it doesn't really play well with other async concepts.

Observable wrapping

We have just learned in Observable Anatomy that the key operators `next()` , `error()` and `complete` is what makes our Observable tick, if we define it ourselves. We have also learned that these methods triggers a corresponding callback on our subscription.

Wrapping something in an observable means we take something that is NOT an Observable and turn it into one, so it can play nice with other Observables. It also means that it can now use Operators.

Wrapping an ajax call

```
import { Observable } from 'rxjs';

let stream = Observable.create(observer => {
  let request = new XMLHttpRequest();

  request.open('GET', 'url');
  request.onload = () => {
    if (request.status === 200) {
      observer.next(request.response);
      observer.complete();
    } else {
      observer.error('error happened');
    }
  };

  request.onerror = () => {
    observer.error('error happened');
  };
  request.send();
});

stream.subscribe(data => console.log(data));
```

Three things we need to do here `emit data` , `handle errors` and `close the stream`

Emit the data

```
if(request.status === 200) {
  observer.next( request.response )  // emit data
}
```

Handle potential errors

```
else {
  observer.error('error happened');
}
```

and

```
request.onerror = () => {
  observer.error('error happened')
}
```

Close the stream

```
if(request.status === 200) {
  observer.next( request.response )
  observer.complete()  // close stream, as we don't expect more data
}
```

Wrapping a speech audio API

```
console.clear();

const { Observable } = Rx;

const speechRecognition$ = new Observable(observer => {
  const speech = new webkitSpeechRecognition();

  speech.onresult = event => {
    observer.next(event);
    observer.complete();
  };

  speech.start();

  return () => {
    speech.stop();
  };
});

const say = text =>
  new Observable(observer => {
    const utterance = new SpeechSynthesisUtterance(text);
    utterance.onend = e => {
      observer.next(e);
```

```
      observer.complete();
    };
    speechSynthesis.speak(utterance);
  });

const button = document.querySelector('button');

const heyClick$ = Observable.fromEvent(button, 'click');

heyClick$
  .switchMap(e => speechRecognition$)
  .map(e => e.results[0][0].transcript)
  .map(text => {
    switch (text) {
      case 'I want':
        return 'candy';
      case 'hi':
      case 'ice ice':
        return 'baby';
      case 'hello':
        return 'Is it me you are looking for';
      case 'make me a sandwich':
      case 'get me a sandwich':
        return 'do it yo damn self';
      case 'why are you being so sexist':
        return 'you made me that way';
      default:
        return `I don't understand: "${text}"`;
    }
  })
  .concatMap(say)
  .subscribe(e => console.log(e));
```

Speech recognition stream

This activates the microphone in the browser and records us

```
const speechRecognition$ = new Observable(observer => {
  const speech = new webkitSpeechRecognition();

  speech.onresult = event => {
    observer.next(event);
    observer.complete();
  };

  speech.start();

  return () => {
    speech.stop();
  };
});
```

This essentially sets up the speech recognition API. We wait for one response and after that we complete the stream, much like the first example with AJAX.

Note also that a function is defined for cleanup

```
return () => {
  speech.stop();
}
```

so that we can call `speechRecognition.unsubscribe()` to clean up resources

Speech synthesis utterance, say

This is responsible for uttering what you want it to utter (say).

```
const say = (text) => new Observable(observer => {
  const utterance = new SpeechSynthesisUtterance(text);
  utterance.onend = (e) => {
    observer.next(e);
    observer.complete();
  };
  speechSynthesis.speak(utterance);
});
```

main stream hey$

```
heyClick$
  .switchMap(e => speechRecognition$)
  .map(e => e.results[0][0].transcript)
  .map(text => {
    switch (text) {
      case 'I want':
        return 'candy';
      case 'hi':
      case 'ice ice':
        return 'baby';
      case 'hello':
        return 'Is it me you are looking for';
      case 'make me a sandwich':
      case 'get me a sandwich':
        return 'do it yo damn self';
      case 'why are you being so sexist':
        return 'you made me that way';
      default:
        return `I don't understand: "${text}"`;
    }
  })
  .concatMap(say)
  .subscribe(e => console.log(e));
```

Logic should be read as follows `heyClick$` is activated on a click on a button. `speechRecognition` is listening for what we say and sends that result into `heyClick$` where the switching logic determines an appropriate response that is uttered by `say` Observable.

all credit due to @ladyleet and @benlesh

Summary

One easier Ajax wrapping and one a little more advanced Speech API has been wrapped into an Observable. The mechanics are still the same though: 1) where data is emitted, add a call to `next()` 2) if there is NO more data to emit call `complete` 3) if there is a need for it, define a function that can be called upon `unsubscribe()` 4) Handle errors through calling `.error()` in the appropriate place. (only done in the first example)

Hot n Cold Observables

There are hot and cold observables. Let's talk about what a cold observable is. In a cold observable two subscribers get their own copies of values like so:

```
// cold observable example
import { of } from 'rxjs';

let stream$ = of(1,2,3);
//subscriber 1: 1,2,3
stream.subscribe(
   data => console.log(data),
   err => console.error(err),
   () => console.log('completed')
)

//subscriber 2: 1,2,3
stream.subscribe(
   data => console.log(data),
   err => console.error(err),
   () => console.log('completed')
)
```

In a hot observable a subscriber receives values when it starts to subscribe, it is however more like a live streaming in football, if you start subscribing 5 minutes in the game, you will have missed the first 5 minutes of action and you start receiving data from that moment on:

```
import { interval } from 'rxjs';
import { take } from 'rxjs/operators';

let liveStreaming$ = interval(1000).pipe(
  take(5)
);

liveStreaming$.subscribe(
  data => console.log('subscriber from first minute')
  err => console.log(err),
  () => console.log('completed')
)

setTimeout(() => {
   liveStreaming$.subscribe(
  data => console.log('subscriber from 2nd minute')
  err => console.log(err),
  () => console.log('completed')
)
},2000)
```

From cold to hot - Katy Perry mode

In the example above it isn't really hot, as a matter of fact both subscribers of the values will each receive `0,1,2,3,4`. As this is the live streaming of a football game it doesn't really act like we want it to, so how to fix it?

Two components are needed to make something go from cold to hot. `publish()` and `connect()`.

```
import { interval } from 'rxjs';
import { take, publish } from 'rxjs/operators';

let publisher$ =
interval(1000).pipe(
  take(5),
  publish()
);


publisher$.subscribe(
  data => console.log('subscriber from first minute',data),
  err => console.log(err),
  () => console.log('completed')
)

setTimeout(() => {
    publisher$.subscribe(
        data => console.log('subscriber from 2nd minute', data),
        err => console.log(err),
        () => console.log('completed')
    )
}, 3000)


publisher$.connect();
```

In this case we see that the output for the first stream that starts to subscribe straight away is `0,1,2,3,4` whereas the second stream is emitting `3,4`. It's clear it matters when the subscription happens.

Warm observables

There is another type of observables that acts a lot like a hot observable but is in a way *lazy*. What I mean with this is that they are essentially not emitting any values until a subscriber arrives. Let's compare a hot and a warm observable

hot observable

```
import { interval } from 'rxjs';
import { take, publish } from 'rxjs/operators';

let stream$ =
interval(1000).pipe(
  take(4),
  publish()
);

stream$.connect();

setTimeout(() => {
  stream$.subscribe(data => console.log(data))
}, 2000);
```

Here we can see that the hot observable will loose the first value being emitted as the subscribe arrives late to the party.

Let's contrast this to our warm observable

warm observable

```
import { interval } from 'rxjs';
import { take, publish, refCount } from 'rxjs/operators';

let obs = interval(1000).pipe(
  take(3),
  publish(),
  refCount()
);

setTimeout(() => {
  obs.subscribe(data => console.log('sub1', data));
},1100)

setTimeout(() => {
  obs.subscribe(data => console.log('sub2', data));
},2100)
```

The `refCount()` operator ensures this observable becomes warm, i.e no values are emitted until `sub1` subscribes. `sub2` on the other hand arrives late to the party, i.e that subscription receives the value its currently on and not the values from the beginning.

So an output from this is

```
sub1 : 0
sub1 : 1
sub2 : 1
sub1 : 2
sub2 : 2
```

This shows the following, first subscriber starts from 0. Had it been hot it would have started at a higher number, i/e it would have been late to the party. When the second subscriber arrives it doesn't get 0 but rather 1 as the first number showing it has indeed become hot.

Naturally hot observables

Generally something is considered hot if the values are emitted straight away without the need for a subscriber to be present. A naturally occuring example of a hot observable is `mousemove`. Most other hot observables are the result of cold observables being turned hot by using `publish()` and `connect()` or by using the `share()` operator.

Sharing

Sharing means using a useful operator called `share()`. Imagine you have the following normal cold observable case :

```
import { Observable } from 'rxjs';

let stream$ = Observable.create((observer) => {
    observer.next( 1 );
    observer.next( 2 );
    observer.next( 3 );
    observer.complete();
}).share()

stream$.subscribe(
    (data) => console.log('subscriber 1', data),
    err => console.error(err),
    () => console.log('completed')
);
stream$.subscribe(
    (data) => console.log('subscriber 2', data),
    err => console.error(err),
    () => console.log('completed')
);
```

If you set a breakpoint on `observer.next(1)` you will notice that it's being hit twice, once for every subscriber. This is the behaviour we expect from a cold observable. Sharing operator is a different way of turning something into a hot observable, in fact it not only turns something hot under the right conditions but it falls back to being a cold observable under certain conditions. So what are these conditions ?

1) **Created as hot Observable** : An Observable has not completed when a new subscription comes and subscribers > 0

2) **Created as Cold Observable** Number of subscribers becomes 0 before a new subscription takes place. I.e a scenario where one or more subscriptions exist for a time but is being unsubscribed before a new one has a chance to happen

3) **Created as Cold Observable** when an Observable completed before a new subscription

Bottom line here is an *active* Observable producing values still and have at least one preexisting subscriber. We can see that the Observable in case 1) is dormant before a second subscriber happens and it suddenly becomes hot on the second subscriber and thereby starts sharing the data where it is.

Operators

Operators are what gives Observables their power. Without them they are nothing. There are close to 60+ operators

Let's look at some:

of

```
import { of } from 'rxjs';

let stream$ = of(1,2,3,4,5);
```

Right here we are using a creation type operator to create an observable for us. This is synchronous in nature and outputs the values as soon as possible. Essentially it lets you specify what values to emit in a comma separated list.

tap

```
import { of } from 'rxjs';
import { tap } from 'rxjs/operators';

let stream$ = of(1,2,3).pipe(
  tap((value) => {
    console.log('emits every value')
  })
);
```

This is a very handy operator as it is used for debugging of your Observable.

filter

```
import { of } from 'rxjs';
import { filter } from 'rxjs/operators';

let stream$ = of(1,2,3,4,5).pipe(
  filter((value) => {
    return value % 2 === 0;
  })
)

// 2,4
```

So this stops certain values from being emitted

Notice however that I can add the `do` operator in a handy place and can still investigate all the values

```
import { of } from 'rxjs';
import { tap, filter } from 'rxjs/operators';

let stream$ = of(1,2,3,4,5).pipe(
  tap((value) => {
    console.log('tap',value)
  }),
  filter((value) => {
    return value % 2 === 0;
  })
);

stream$.subscribe((value) => {
  console.log('value', value)
})

//  tap: 1,tap : 2, tap : 3, tap : 4, tap: 5
// value : 2, 4
```

Marble Diagrams

A marble diagram is a graphical representation of applying one or more operators to x number of streams. It can look like this :

```
---v-----v---->
-------v------>
operator
---r---r------>
```

The whole point is to make it easier to understand what the operator does. Most operators in Rxjs is covered at Rx Marbles

Operators and Ajax

There is an `ajax` operator on the Rx object.

Using the ajax() operator

index.html

```
<html>
  <body>
    <div id="result">
    </div>
    <script src="https://unpkg.com/@reactivex/rxjs@5.0.1/dist/global/Rx.js"></script>
        <script src="app.js"></script>
  </body>
</html>
```

app.js

```
import { ajax } from 'rxjs/ajax';
import { map } from 'rxjs/operators';

let person$ = .ajax({
      url : 'http://swapi.co/api/people/1',
      crossDomain: true,
      createXHR: function () {
        return new XMLHttpRequest();
     }
    })
  .pipe(
    map(e => e.response)
  );

const subscription = person$
  .subscribe(res => {
    let element = document.getElementById('result');
    element.innerHTML = res.name
    console.log(res)
  });
```

A little GOTCHA from this is how we call the `ajax()` operator, we obviously specify a bunch of stuff other thant the `url` property. The reason for this is that the `ajax` operator does the following :

> default factory of XHR in ajaxObservable sets withCredentials to true by default

So we give at a custom factory and it works. I understand this is an issue that is currently looked upon

Using fetch API

```
const fetchSubscription = Rx.Observable
.from(fetch('http://swapi.co/api/people/1'))
.flatMap((res) => Rx.Observable.from(res.json()) )
.subscribe((fetchRes) => {
    console.log('fetch sub', fetchRes);
})
```

So a couple of things here happens worth mentioning

- fetch api is promised base, however using `.from()` Rxjs allows us to insert promise as a parameter and converts it to an Observable.
- BUT the result coming back is a response object that we need to convert to Json. Calling `json()` will do that for you but that operation returns a Promise. So we need to use another `from()` operator. But creating an Observable inside an observable creates a list of observables and we can't have that, we want Json. So we use an operator called `flatMap()` to fix that. Read more on `flatMap()` here

And finally we get the Json we expect, no issues with CORS but a little more to write.

Observable in an Observable

Quite often you come to a point where you start with one type of Observable and you want it to turn into something else.

Example

```
import { of } from 'rxjs';
import { ajax } from 'rxjs/ajax';
import { flatMap, map } from 'rxjs/operators';

let stream$ = of(1,2,3)
.pipe(
  flatMap((val) => {
    return of(val).pipe(
        ajax({ url : url + })).pipe(
          map((e) => e.response)
        )
    );
}

stream.subscribe((val) => console.log(val))

// { id : 1, name : 'Darth Vader' },
// { id : 2, name : 'Emperor Palpatine' },
// { id : 3, name : 'Luke Skywalker' }
```

So here we have a case of starting with values 1,2,3 and wanting those to lead up to an ajax call each

--1------2-----3------> --json-- json--json -->

The reason for us NOT doing it like this with a `.map()` operator

```
import { of } from 'rxjs';
import { ajax } from 'rxjs/ajax';
import { map } from 'rxjs/operators';

let stream$ = of(1,2,3).pipe(
  map((val) => {
    return of(val)
      .pipe(
        ajax({ url : url + }) )
        .pipe(
          map((e) => e.response )
        )
      )
  }
);
```

is that it would not give the result we want instead the result would be:

```
// Observable, Observable, Observable
```

because we have created a list of observables, so three different streams. The `flatMap()` operator however is able to flatten these three streams into one stream called a `metastream`. There is however another interesting operator that we should be using when dealing with ajax generally and it's called `switchMap()`. Read more about it here [Cascading calls]

Operators construction

create

When you are starting out or you just want to test something you tend to start out with the `create()` operator. This takes a function with an `observer` as a parameter. This has been mentioned in previous sections such as Observable Wrapping. The siignature looks like the following

```
Observable.create([fn])
```

And an example looks like:

```
import { Observable } from 'rxjs';

Observable.create(observer => {
  observer.next( 1 );
})
```

range

Signature

```
import { range } from 'rxjs';

range([start],[count])
```

Example

```
import { range } from 'rxjs';


let stream$ = range(1,3)

// emits 1,2,3
```

Operators conversion

The whole point with this category is to show how easy it is to create Observables from something so they can play nice with the operators and well whatever construct you come from enables rich composition.

from

In Rxjs4 there existed a bunch of operators with resembling names such as `fromArray()`, `from()`, `fromPromise()` etc. All these `from()` operators are now just `from()`. Let's look at some examples:

old fromArray

```
import { from } from 'rxjs';

from([2,3,4,5])
```

old fromPromise

```
import { from } from 'rxjs';

from(new Promise(resolve, reject) => {
  // do async work
  resolve( data )
})
```

of

The `of` operator takes x number of arguments so you can call it with one arguments as well as 10 arguments like so:

```
import { of } from 'rxjs';


of(1,2);
of(1,2,3,4);
```

to

There also exist a bunch of operators that allows you go the other way, i.e leave the wonderful world of observables and go back to a more primitive state like :

```
import { of } from 'rxjs';

let promise = of(1,2).toPromise();
promise.then(data => console.log('Promise', data));
```

Operators combination

There are many operators out there that allows you to combine the values from 2 or more source, they act a bit differently though and its important to know the difference.

combineLatest

The signature on this one is:

```
import { combineLatest } from 'rxjs';

combineLatest([ source_1, ...  source_n])
```

```
import { interval } from 'rxjs';
import { map, take } from 'rxjs/operators';

let source1 = interval(100).pipe(
  map( val => "source1 " + val ),
  take(5)
);

let source2 = interval(50).pipe(
  map( val => "source2 " + val),
  take(2)
);

let stream$ = combineLatest(
    source1,
    source2
);

stream$.subscribe(data => console.log(data));

// emits source1: 0, source2 : 0 |  source1 : 0, source2 : 1 | source1 : 1, source2 :
1, etc
```

What this does is to essentially take the latest response from each `source` and return it as an array of x number of elements. One element per source.

As you can see source2 stops emitting after 2 values but is able to keep sending the latest emitted.

Business case

The business case is when you are interested in the very latest from each source and past values is of less interest, and of course you have more than one source that you want to combine.

concat

The signature is :

```
Observable([ source_1,... sournce_n ])
```

Looking at the following data, it's easy to think that it should care when data is emitted:

```
import { interval, concat } from 'rxjs';

let source1 = interval(100).pipe(
  map( val => "source1 " + val ),
  take(5)
);

let source2 = interval(50).pipe(
  map( val => "source2 " + val ),
  take(2)
);


let stream$ = concat(
    source1,
    source2
);

stream$.subscribe( data => console.log('Concat ' + data));

// source1 : 0, source1 : 1, source1 : 2, source1 : 3, source1 : 4
// source2 : 0, source2 : 1
```

The result however is that the resulting observable takes all the values from the first source and emits those first then it takes all the values from source 2, so order in which the source go into `concat()` operator matters.

So if you have a case where a source somehow should be prioritized then this is the operator for you.

merge

This operator enables you two merge several streams into one.

```
import { merge } from 'rxjs';

let merged$ = merge(
  Rx.Observable.of(1).delay(500),
  Rx.Observable.of(3,2,5)
);

let observer = {
  next : data => console.log(data)
}

merged$.subscribe(observer);
```

Point with is operator is to combine several streams and as you can see above any time operators such as `delay()` is respected.

zip

```
import { zip, of } from 'rxjs';

let stream$ = zip(
  Promise.resolve(1),
  of(2,3,4),
  of(7)
);


stream$.subscribe(observer);
```

Gives us `1,2,7`

Let's look at another example

```
import { zip, of } from 'rxjs';

let stream$ = zip(
  of(1,5),
  of(2,3,4),
  of(7,9)
);

stream$.subscribe(observer);
```

Gives us `1,2,7` and `5,3,9` so it joins values on column basis. It will act on the least common denominator which in this case is 2. The `4` is ignored in the `2,3,4` sequence. As you can see from the first example it's also possible to mix different async concepts such as Promises with Observables because interval conversion happens.

Business case

If you really care about what different sources emitted at a certain position. Let's say the 2nd response from all your sources then `zip()` is your operator.

Operators matchematical

max

```
import { of } from 'rxjs';
import { max } from 'rxjs/operators';

let stream$ = of(5,4,7,-1).pipe(
  max()
)
```

This emits `7`. It's obvious what this operator does, it gives us just one value, the max. There is different ways of calling it though. You can give it a comparer:

```
import { of } from 'rxjs';

function comparer(x,y) {
    if( x > y ) {
        return 1;
    } else if( x < y ) {
        return -1;
    } else return 0;
}

let stream$ = of(5,4,7,-1).pipe(
  max(comparer)
);
```

In this case we define a `comparer` which runs a sort algorithm under the hood and all we have to do is to help it determine when something is *larger than*, *equal* or *smaller than*. Or with an object the idea is the same:

```
import { of } from 'rxjs';
import { max } from 'rxjs/operators'

function comparer(x,y) {
  if( x.age > y.age ) {
    return 1;
  } else if( x.age < y.age ) {
    return -1;
  } else return 0;
}

let stream$ = of(
  { name : 'chris', age : 37 },
  { name : 'chross', age : 32 })
.pipe(
  max(comparer)
);
```

Because we tell it in the `comparer` what property to compare we are left with the first entry as result.

min

Min is pretty much identical to `max()` operator but returns the opposite value, the smallest.

sum

`sum()` as operator has seized to exist but we can use `reduce()` for that instead like so:

```
import { of } from 'rxjs';
import { reduce } from 'rxjs/operators';

let stream$ = of(1,2,3,4).pipe(
  reduce((accumulated, current) => accumulated + current )
)
```

This can be applied to objects as well as long as we define what the `reduce()` function should do, like so:

```
import { of } from 'rxjs';
import { reduce } from 'rxjs/operators';

let objectStream$ = of(
  { name : 'chris' },
  { age : 11 }
).pipe(
  reduce(
    (acc,curr) => Object.assign({}, acc,curr )
  )
);
```

This will concatenate the object parts into an object.

average

The `average()` operator isn't there anymore in Rxjs5 but you can still achieve the same thing with a `reduce()`

```
import { of } from 'rxjs';
import { map } from 'rxjs/operators';

let stream$ = of( 3, 6 ,9 ).pipe(
  map( x => { return { sum : x, counter : 1 } }),
  reduce( (acc,curr) => {
   return Object.assign(
     {},
     acc,
     { sum : acc.sum + curr.sum ,counter : acc.counter + 1  }
   ),
   map( x => x.sum / x.counter )

});
```

I admit it, this one hurted my head a little, once you crack the initial `map()` call the `reduce()` is pretty simple, and `Object.assign()` is a nice companion as usual.

Operators time

This is not an easy topic. There are many areas of application here, either you might want to synchronize responses from APIS or you might want to deal with other types of streams such as events like clicks or keyup in a UI.

There are plenty of operators dealing with time in some way such as `delay` `debounce` `throttle` `interval` etc.

This is not an easy topic. There are many areas of application here, either you might want to synchronize responses from APIS or you might want to deal with other types of streams such as events like clicks or keyup in a UI.

interval

This operator is used to construct an Observable and essentially what it does is to pump values at regular interval, signature:

```
import { interval } from 'rxjs';

interval([ms])
```

Example usage:

```
import { interval } from 'rxjs';

interval(100)

// generates forever
```

Because this one will generate values forever you tend to want to combine it with the `take()` operator that limits the amount of values to generate before calling it quits so that would look like :

```
import { interval } from 'rxjs';

interval(1000).take(3)

// generates 1,2,3
```

timer

Timer is an interesting one as it can act in several ways depending on how you call it. It's signature is

```
import { timer } from 'rxjs';

timer([initial delay],[thereafter])
```

However only the initial args is mandatory so depending on the number of args used these are the different types that exist because of it.

one-off

```
import { timer } from 'rxjs';

let stream$ = timer(1000);

stream$.subscribe(data => console.log(data));

// generates 0 after 1 sec
```

This becomes a one-ff as we don't define when the next value should happen.

with 2nd arg specified

```
import { timer } from 'rxjs';
import { take } from 'rxjs/operators';

let moreThanOne$ = timer(2000, 500).pipe(
  take(3)
);

moreThanOne$.subscribe(data => console.log('timer with args', data));

// generate 0 after 2 sec and thereafter 1 after 500ms and 2 after additional 500ms
```

So this one is more flexible and keeps emitting values according to 2nd argument.

delay

`delay()` is an operator that delays every value being emitted Quite simply it works like this :

```
import { interval } from 'rxjs';
import { take, delay } from 'rxjs/operators';

var start = new Date();
let stream$ = interval(500).pipe(
  take(3)
);

stream$.pipe(
  delay(300)
)
.subscribe((x) => {
  console.log('val',x);
  console.log( new Date() - start );
})

//0 800ms, 1 1300ms,2 1800ms
```

Business case

Delay can be used in a multitude of places but one such good case is when handling errors especially if we are dealing with `shaky connections` and want it to retry the whole stream after x miliseconds:

Read more in chapter Error handling

sample

I usually think of this scenario as *talk to the hand*. What I mean by that is that events are only fired at specific points

Business case

So the ability to ignore events for x miliseconds is pretty useful. Imagine a save button being repeatedly pushed. Wouldn't it be nice to only act after x miliseconds and ignore the other pushes ?

```
import { fromEvent } from 'rxjs';
import { sampleTime } from 'rxjs/operators';

const btn = document.getElementById('btnIgnore');
var start = new Date();

const input$ = fromEvent(btn, 'click').pipe(
  sampleTime(2000)
);

input$.subscribe(val => {
  console.log(val, new Date() - start);
});
```

The code above does just that.

debounceTime

So `debounceTime()` is an operator that tells you: I will not emit the data all the time but at certain intervals.

Business case

Debounce is a known concept especially when you type keys on a keyboard. It's a way of saying we don't care about every keyup but once you stop typing for a while we should care. That, is how you would normally start an auto complete. Say your user hasn't typed for x miliseconds that probably means we should be doing an ajax call and retrieve a result.

```
import { fromEvent } from 'rxjs';
import { map, debounceTime } from 'rxjs/operators';

const input = document.getElementById('input');

const example = fromEvent(input, 'keyup').pipe(
  map(i => i.currentTarget.value)
);

//wait 0.5s, between keyups, throw away all other values
const debouncedInput = example.pipe(
  debounceTime(500)
);

const subscribe = debouncedInput.subscribe(val => {
  console.log(`Debounced Input: ${val}`);
});
```

The following only outputs a value, from our input field, after you stopped typing for 500ms, then it's worth reporting about it, i.e emit a value.

throttleTime

TODO

buffer

This operator has the ability to record x number of emitted values before it outputs its values, this one comes with one or two input parameters.

```
import { buffer } from 'rxjs/operators';



buffer( whenToReleaseValuesStartObservable )

or

buffer(
  whenToReleaseValuesStartObservable,
  whenToReleaseValuesEndObservable
)
```

So what does this mean? It means given we have for example a click of streams we can cut it into nice little pieces where every piece is equally long. Using the first version with one parameter we can give it a time argument, let's say 500 ms. So something emits values for 500ms then the values are emitted and another Observable is started, and the old one is abandoned. It's much like using a stopwatch and record for 500 ms at a time. Example :

```
import { interval } from 'rxjs';
import { take, buffer } from 'rxjs/operators';

let scissor$ = interval(500);

let emitter$ = interval(100).pipe(
  take(10),
  .buffer( scissor$ )
) // output 10 values in total


// [0,1,2,3,4] 500ms [5,6,7,8,9]
```

Marble diagram

```
--- c --- c - c --- >
-------| ------- |- >
Resulting stream is :
------ r ------- r r  -- >
```

Business case

So whats the business case for this one? `double click`, it's obviously easy to react on a `single click` but what if you only want to perform an action on a `double click` or `triple click`, how would you write code to handle that? You would probably start with something looking like this :

```

$('#btn').bind('click', () => {
  if(!start) {
    start = timer.start();
  }
  timePassedSinceLastClickInMs = now - start;
  if (timePassedSinceLastClickInMs < 250) {
    console.log('double click');
  } else {
     console.log('single click')
  }

  start = timer.start();
})
```

Look at the above as an attempt at pseudo code. The point is that you need to keep track of a bunch of variables of how much time has passed between clicks. This is not nice looking code, lacks elegance

Model in Rxjs

By now we know Rxjs is all about streams and modeling values over time. Clicks are no different, they happen over time.

```
---- c ---- c ----- c ----- >
```

We however care about the clicks when they appear close together in time, i.e as double or triple clicks like so :

```
--- c - c ------ c -- c -- c ----- c
```

From the above stream you should be able to deduce that a `double click`, one `triple click` and one `single click` happened.

So let's say I do, then what? You want the stream to group itself nicely so it tells us about this, i.e it needs to emit these clicks as a group. `filter()` as an operator lets us do just that. If we define let's say 300ms is a long enough time to collect events on, we can slice up our time from 0 to forever in chunks of 300 ms with the following code:

```
import { fromEvent, interval } from 'rxjs';
import { buffer } from 'rxjs/operators';

let clicks$ = fromEvent(document.getElementById('btn'), 'click');

let scissor$ = interval(300);

clicks$.pipe(
  buffer( scissor$ )
    //.filter( (clicks) => clicks.length >=2 )
    .subscribe((value) => {
      if(value.length === 1) {
        console.log('single click')
      }
      else if(value.length === 2) {
        console.log('double click')
      }
      else if(value.length === 3) {
        console.log('triple click')
      }
    })
);
```

Read the code in the following way, the buffer stream, `clicks$` will emit its values every 300ms, 300 ms is decided by `scissor$` stream. So the `scissor$` stream is the scissor, if you will, that cuts up our click stream and voila we have an elegant `double click` approach. As you can see the above code captures all types of clicks but by uncommenting the `filter()` operation we get only `double clicks` and `triple clicks`.

`filter()` operator can be used for other purposes as well like recording what happened in a UI over time and replay it for the user, only your imagination limits what it can be used for.

Operators Grouping

buffer

The signature of `buffer()` operator is :

```
buffer([breakObservable])
```

Buffer itself means we wait with emitting any values until the `breakObservable` happens. An example of that is the following:

```
import { timer, interval } from 'rxjs';
import { buffer } from 'rxjs/operators';

let breakWhen$ = timer(1000);

let stream$ = interval(200).pipe(
  buffer( breakWhen$ )
);

stream$.subscribe((data) => console.log( 'values',data ));
```

In this case the values 0,1,2,3 is emitted all at once.

Business case

Auto complete
The most obvious case when dealing with the `buffer()` operator is an `auto complete`. But how does `auto complete` work? Let's look at it in steps

- user enter keys
- search is made base on those keystrokes The important thing though is that the search itself is carried out as you are typing, either it's carried out because you typed x number of characters or the more common approach is to let you finish typing and do the search, you could be editing as you type. So let's take our first step into such a solution:

```
import { fromEvent, timer } from 'rxjs';
import { debounceTime, map, buffer } from 'rxjs/operators';


let input = document.getElementById('example');
let input$  = fromEvent( input, 'keyup' )

let breakWhen$ = timer(1000);
let debounceBreak$ = input$.pipe(
  debounceTime( 2000 )
);

let stream$ = input$.pipe(
  map( ev => ev.key )
  buffer( debounceBreak$ )
);

stream$.subscribe((data) => console.log( 'values',data ));
```

We capture `keyup` events. We also use a `debounce()` operator that essentially says; I will emit values once you stopped typing for x miliseconds. This solution is just a first step on the way however as it is reporting the exact keys being typed. A better solution would be to capture the input element's actual content and also to perform an ajax call, so let's look at a more refined solution:

```
import { fromEvent, timer } from 'rxjs';
import { debounceTime, map, buffer, switchMap } from 'rxjs/operators';

let input = document.getElementById('example');
let input$  = fromEvent(input, 'keyup');

let breakWhen$ = timer(1000);
let debounceBreak$ = input$.pipe(
  debounceTime( 2000 )
);

let stream$ = input$.pipe(
  map( ev => ev.key),
  buffer(debounceBreak$),
  switchMap((allTypedKeys) => {
    // do ajax
    console.log('Everything that happened during 2 sec', allTypedKeys)
    return of('ajax based on ' + input.value);
  });
);

stream$.subscribe((data) => console.log( 'values',data ));
```

Let's call this one `auto complete on steroids`. The reason for the name is that we save every single interaction the user does before finally deciding on the final input that should become an Ajax call. So a result from the above could look like the following:

```
// from switchMap
Everything that happened during 2 sec ["a", "a", "a", "Backspace", "Backspace", "Backs
pace", "Backspace", "b", "b", "Backspace", "Backspace", "Backspace", "f", "g", "h", "f"
, "h", "g"]

// in the subscribe(fnValue)
app-buffer.js:31 values ajax based on fghfgh
```

As you can see we could potentially store a whole lot more about a user than just the fact that they made an auto complete search, we can store how they type and that may or may not be interesting..

Double click

In the example above I've showed how it could be interesting to capture groups of keys but another group of UI events of possible interests are mouse clicks, namely for capturing single, double or triple clicks. This is quite inelegant code to write if not in Rxjs but with it, it's a breeze:

```
import { fromEvent } from 'rxjs';
import { debounceTime, buffer, map, filter } from 'rxjs/operators';

let btn = document.getElementById('btn2');
let btn$  = fromEvent(btn, 'click')

let debounceMouseBreak$ = btn$.pipe(
  debounceTime(300)
);

let btnBuffered$ = btn$.pipe(
  buffer( debounceMouseBreak$),
  map( array => array.length),
  filter( count => count >= 2)
)
;

btnBuffered$.subscribe((data) => console.log( 'values',data ));
```

Thanks to the `debounce()` operator we are able to express wait for 300ms before emitting anything. This is quite few lines and it's easy for us to decide what our filter should look like.

bufferTime

The signature of `bufferTime()` is

```
bufferTime([ms])
```

The idea is to record everything that happens during that time slice and output all the values. Below is an example of recording all activities on an input in 1 second time slices.

```
import { fromEvent } from 'rxjs';
import { bufferTime } from 'rxjs/operators';

let input = document.getElementById('example');
let input$  = fromEvent( input, 'input').pipe(
  bufferTime(1000)
);

input$.subscribe((data) => console.log('all inputs in 1 sec', data));
```

In this case you will get an output looking like:

```
all inputs in 1 sec [ Event, Event... ]
```

Not so usable maybe so we probably need to make it nice with a `filter()` to see what was actually typed, like so:

```
import { fromEvent } from 'rxjs';
import { map, bufferTime } from 'rxjs/operators';

let input = document.getElementById('example');
let input$  = fromEvent( input, 'keyup').pipe(
  map( ev => ev.key),
  bufferTime(1000)
);

input$.subscribe((data) => console.log('all inputs in 1 sec', data));
```

Also note I changed event to `keyup`. Now we are able to see all `keyup` events that happened for a sec.

Business case

The example above could be quite usable if you want to record what another user on the site is doing and want to replay all the interactions they ever did or if they started to type and you want to send this info over a socket. The last is something of a standard functionality nowadays that you see a person typing on the other end. So there are definitely use cases for this.

groupBy

TODO

Error handling

There are two major approaches how to handle errors in streams. You can retry your stream and how it eventually will work or you can take the error and transform it.

Retry - how bout now?

This approach makes sense when you believe the error is temporary for some reason. Usually *shaky connections* is a good candidate for this. With a *shaky connection* the endpoint might be there to answer like for example every 5th time you try. Point is the first time you try it *might* fail, but retrying x times, with a certain time between attempts, will lead to the endpoint finally answering.

retry

The `retry()` operator lets us retry the whole stream, value for value x number of times having a signature like this :

```
retry([times])
```

The important thing to note with the `retry()` operator is that it delays when the error callback is being called. Given the following code the error callback is being hit straight away:

```
import { of } from 'rxjs';
import { map } from 'rxjs/operators';

let stream$ = of(1,2,3).pipe(
  map(value => {
    if(value > 2) { throw 'error' }
  })
);

stream$.subscribe(
   data => console.log(data),
   err => console.log(err)
)
```

The stream effectively dies when the error callback is being hit and this is where the `rety()` operator comes in. By appending it like so:

```
import { of } from 'rxjs';

let stream$ = of(1,2,3).pipe(
  map(value => {
   if(value > 2) { throw 'error' }
  }),
  retry(5)
)
```

This will run the sequence of values 5 more times before finally giving up and hitting the error callback. However in this case, the way the code is written, it will just generate `1,2` five times. So our code isn't really utilizing the operator to its fullest potential. What you probably want is to be able to change something between attempts. Imagine your observable looked like this instead:

```
import { of } from 'rxjs';

let urlsToHit$ = of(url, url2, url3);
```

In this its clearly so that an endpoint might have answered badly or not at all on your first attempt and it makes sense to retry them x number of times.

However in the case of ajax calls, and imagining our business case is *shaky connections* it makes no sense to do the retry immediately so we have to look elsewhere for a better operator, we need to look to `retryWhen()`

retryWhen

A `retryWhen()` operator gives us the chance to operate on our stream and handle it appropriately

```
retryWhen( stream => {
   // return it in a better condition, hopefully
})
```

Lets' write a piece of naive code for a second

```
import { of } from 'rxjs';
import { map } from 'rxjs/operators';

let values$ = of( 1,2,3,4 ).pipe(
  map(val => {
    if(val === 2) {
      throw 'err';
    }
    else return val;
  }),
  retryWhen( stream => {
    return stream;
  })
);

values$.subscribe(
  data => console.log('Retry when - data',data),
  err => console.error('Retry when - Err',err)
)
```

The way it's written it will return `1` until we run out of memory cause the algorithm will always crash on the value `2` and will keep retrying the stream forever, due to our lack of end condition. What we need to do is to somehow say that the error is fixed. If the stream were trying to hit urls instead of emitting numbers a responding endpoint would be the fix but in this case we have to write something like this:

```
import { interval } from 'rxjs';
import { take, map } from 'rxjs/operators';

let values$ = interval(1000).pipe(
  take(5)
)

let errorFixed = false;

values$.pipe(
  map((val) => {
   if(errorFixed) {
     return val;
   }
   else if( val > 0 && val % 2 === 0) {
     errorFixed = true;
     throw { error : 'error' };
   } else {
      return val;
   }
 }),
 retryWhen((err) => {
   console.log('retrying the entire sequence');
   return err;
 })
)
.subscribe((val) => { console.log('value',val) })

// 0 1 'wait 200ms' retrying the whole sequence 0 1 2 3 4
```

This however resembles a lot of what we did with the `retry()` operator, the code above will just retry once. The real benefit is being to change the stream we return inside the `retryWhen()` namely to involve a delay like this:

```
.retryWhen((err) => {
    console.log('retrying the entire sequence');
    return err.delay(200)
})
```

This ensures there is a 200ms delay before sequence is retried, which in an ajax scenario could be enough for our endpoint to *get it's shit together* and start responding.

GOTCHA

The `delay()` operator is used within the `retryWhen()` to ensure that the retry happens a while later to in this case give the network a chance to recover.

retryWhen with delay and no of times

So far `retry()` operator has been used when we wanted to retry the sequence x times and `retryWhen()` has been used when we wanted to delay the time between attempts, but what if we want both. Can we do that? We can. We need to think about us somehow remembering the number of attempts we have made so far. It's very tempting to introduce an external variable and keep that count, but that's not how we do things the functional way, remember side effects are forbidden. So how do we solve it? There is an operator called `scan()` that will allow us to accumulate values for every iteration. So if you use scan inside of the `retryWhen()` we can track our attempts that way:

```
import { of } from 'rxjs';
import { map, retryWhen } from 'rxjs/operators';

let ATTEMPT_COUNT = 3;
let DELAY = 1000;
let delayWithTimes$ = of(1,2,3).pipe(
  map( val => {
    if(val === 2) {
      throw 'err'
    }
    else {
      return val;
    }
  }),
  retryWhen(e => e.scan((errorCount, err) => {
    if (errorCount >= ATTEMPT_COUNT) {
      throw err;
    }
    return errorCount + 1;
  }, 0).delay(DELAY));
);

delayWithTimes$.subscribe(
  val => console.log('delay and times - val',val),
  err => console.error('delay and times - err',err)
)
```

Transform - nothing to see here folks

This approach is when you get an error and you choose to remake it into a valid Observable.

So lets exemplify this by creating an Observable who's mission in life is to fail miserably

```
import { throwError } from 'rxjs/operators';

let error$ = Rx.Observable.throwError('crash');

error$.subscribe(
  data => console.log( data ),
  err => console.log( err ),
  () => console.log('complete')
)
```

This code will only execute the error callback and NOT reach the complete callback.

Patching it

We can patch this by introducing the `catch()` operator. It is used like this:

```
import { of } from 'rxjs';

let errorPatched$ = error$.catch(
  err => { return of('Patched' + err) }
);

errorPatched$.subscribe((data) => console.log(data) );
```

As you can see `patching it` with `.catch()` and returning a new Observable *fixes* the stream. Question is if that is what you want. Sure the stream survives and reaches completion and can emit any values that happened after the point of crash.

If this is not what you want then maybe the Retry approach above suits you better, you be the judge.

What about multiple streams?

You didn't think it would be that easy did you? Usually when coding Rxjs code you deal with more than one stream and using `catch()` operator approach is great if you know where to place your operator.

```
import { of, merge } from 'rxjs';
import { throwError } from 'rxjs/operators';

let badStream$ = throwError('crash');
let goodStream$ = of(1,2,3,);

let merged$ = merge(
  badStream$,
  goodStream$
);

merged$.subscribe(
  data => console.log(data),
  err => console.error(err),
  () => console.log('merge completed')
)
```

Care to guess what happened? 1) crash + values is emitted + complete 2) crash + values is emitted 3) crash only is emitted

Sadly 3) is what happens. Which means we have virtually no handling of the error.

Lets patch it S we need to patch the error. We do patching with `catch()` operator. Question is where?

Let's try this?

```
import { merge, of } from 'rxjs';

let mergedPatched$ = merge(
  badStream$,
  goodStream$
).catch(err => of(err));

mergedPatched$.subscribe(
  data => console.log(data),
  err => console.error(err),
  () => console.log('patchedMerged completed')
)
```

In this case we get 'crash' and 'patchedMerged completed'. Ok so we reach complete but it still doesn't give us the values from `goodStream$`. So better approach but still not good enough.

Patch it better So adding the `catch()` operator after the `merge()` ensured the stream completed but it wasn't good enough. Let's try to change the placement of `catch()`, pre merge.

```
import { merge, of } from 'rxjs';

let preMergedPatched$ = merge(
  badStream$.catch(err => of(err)),
  goodStream$
).catch(err => of(err));

preMergedPatched$.subscribe(
  data => console.log(data),
  err => console.error(err),
  () => console.log('pre patched merge completed')
)
```

And voila, we get values, our error emits its error message as a new nice Observable and we get completion.

GOTCHA It matters where the `catch()` is placed.

Survival of the fittest

There is another scenario that might be of interest. The above scenario assumes you want everything emitted, error messages, values, everything.

What if that is not the case, what if you only care about values from streams that behave? Let's say thats your case, there is an operator for that `onErrorResumeNext()`

```
import { throwError, of } from 'rxjs';
import { onErrorResumeNext } from 'rxjs/operators';

let secondBadStream$ = throwError('bam');
let gloriaGaynorStream$ = of('I will survive');

let emitSurviving = onErrorResumeNext(
  badStream$,
  secondBadStream$,
  gloriaGaynorStream$
);

emitSurviving.subscribe(
  data => console.log(data),
  err => console.error(err),
  () => console.log('Survival of the fittest, completed')
)
```

The only thing emitted here is 'I will survive' and 'Survival of the fittest, completed'.

Testing

Testing of async code is generally quite tricky. Async code may finish in ms or even minutes. So you need a way to either mock it away completely like for example you do with jasmine.

```
spyOn(service,'method').and.callFake(() => {
    return {
        then : function(resolve, reject){
            resolve('some data')
        }
    }
})
```

or a more shorthand version:

```
spyOn(service,'method').and.callFake(q.when('some data'))
```

Point is you try to avoid the whole timing thing. Rxjs have historically, in `Rxjs 4` provided the approach of using a TestScheduler with its own internal clock, which has enabled you to increment time. This approach have had two flavors :

Approach 1

```
let testScheduler = new TestScheduler();

// my algorithm
let stream$ = Rx.Observable
.interval(1000, testScheduler)
.take(5);



// setting up the test
let result;

stream$.subscribe(data => result = data);


testScheduler.advanceBy(1000);
assert( result === 1 )

testScheduler.advanceBy(1000);
... assert again, etc..
```

This approach was pretty easy to grok. The second approach was using hot observables and a `startSchedule()` method, looking something like this :

```
// setup the thing that outputs data
var input = scheduler.createHotObservable(
    onNext(100, 'abc'),
    onNext(200, 'def'),
    onNext(250, 'ghi'),
    onNext(300, 'pqr'),
    onNext(450, 'xyz'),
    onCompleted(500)
  );

// apply operators to it
var results = scheduler.startScheduler(
    function () {
      return input.buffer(function () {
        return input.debounce(100, scheduler);
      })
      .map(function (b) {
        return b.join(',');
      });
    },
    {
      created: 50,
      subscribed: 150,
      disposed: 600
    }
  );

//assert
collectionAssert.assertEqual(results.messages, [
    onNext(400, 'def,ghi,pqr'),
    onNext(500, 'xyz'),
    onCompleted(500)
  ]);
```

A little harder to read IMO but you still get the idea, you control time because you have a `TestScheduler` that dictates how fast time should pass.

This is all Rxjs 4 and it has changed a bit in Rxjs 5. I should say that what I am about to write down is a bit of a general direction and a moving target so this chapter will be updated, but here goes.

In Rxjs 5 something called `Marble Testing` is used. Yes that is related to Marble Diagram i.e you express your expected input and actual output with graphical symbols.

First time I had a look at the offical docs page I was like *What now with a what now?*. But after writing a few tests myself I came to the conclusion this is a pretty elegant approach.

So I will explain it by showing you code:

```
// setup
const lhsMarble = '-x-y-z';
const expected = '-x-y-z';
const expectedMap = {
    x: 1,
    y: 2,
    z : 3
};

const lhs$ = testScheduler.createHotObservable(lhsMarble, { x: 1, y: 2, z :3 });

const myAlgorithm = ( lhs ) =>
    Rx.Observable
    .from( lhs );

const actual$ = myAlgorithm( lhs$ );

//assert
testScheduler.expectObservable(actual$).toBe(expected, expectedMap);
testScheduler.flush();
```

Let's break it down part by part

Setup

```
const lhsMarble = '-x-y-z';
const expected = '-x-y-z';
const expectedMap = {
    x: 1,
    y: 2,
    z : 3
};

const lhs$ = testScheduler.createHotObservable(lhsMarble, { x: 1, y: 2, z :3 });
```

We essentially create a pattern instruction `-x-y-z` to the method `createHotObservable()` that exist on our `TestScheduler`. This is a factory method that does some heavy lifting for us. Compare this to writing this by yourself, in which case it corresponds to something like:

```
let stream$ = Rx.Observable.create(observer => {
   observer.next(1);
   observer.next(2);
   observer.next(3);
})
```

The reason we don't do it ourselves is that we want the `TestScheduler` to do it, so time passes according to its internal clock. Note also that we define an expected pattern and an expected map:

```
const expected = '-x-y-z';

const expectedMap = {
    x: 1,
    y: 2,
    z : 3
}
```

Thats what we need for the setup, but to make the test run we need to `flush` it so that `TestScheduler` internally can trigger the HotObservable and run an assert. Peeking at `createHotObservable()` method we find that it parses the marble patterns we give it and pushes it to list:

```
// excerpt from createHotObservable
 var messages = TestScheduler.parseMarbles(marbles, values, error);
var subject = new HotObservable_1.HotObservable(messages, this);
this.hotObservables.push(subject);
return subject;
```

Next step is assertion which happens in two steps
1) expectObservable()
2) flush()

The expect call pretty much sets up a subscription to our HotObservable

```
// excerpt from expectObservable()
this.schedule(function () {
    subscription = observable.subscribe(function (x) {
        var value = x;
        // Support Observable-of-Observables
        if (x instanceof Observable_1.Observable) {
            value = _this.materializeInnerObservable(value, _this.frame);
        }
        actual.push({ frame: _this.frame, notification: Notification_1.Notification.cr
eateNext(value) });
    }, function (err) {
        actual.push({ frame: _this.frame, notification: Notification_1.Notification.cr
eateError(err) });
    }, function () {
        actual.push({ frame: _this.frame, notification: Notification_1.Notification.cr
eateComplete() });
    });
}, 0);
```

by defining an internal `schedule()` method and invoking it.
The second part of the assert is the assertion itself:

```
// excerpt from flush()
 while (readyFlushTests.length > 0) {
    var test = readyFlushTests.shift();
    this.assertDeepEqual(test.actual, test.expected);
}
```

It ends up comparing two lists to each other, the `actual` and `expect` list.
It does a deep compare and verifies two things, that the data happened on the correct time `frame` and that the value on that frame is correct. So both lists consist of objects that looks like this:

```
{
  frame : [some number],
  notification : { value : [your value] }
}
```

Both these properties must be equal for the assert to be true.

Doesn't seem that bloody right?

Symbols

I havn't really explained what we looked at with:

```
-a-b-c
```

But it actually means something. `-` means a time frame passed. `a` is just a symbol. So it matters how many `-` you write in actual and expected cause they need to match. Let's look at another test so you get the hang of it and to introduce more symbols:

```
const lhsMarble = '-x-y-z';
const expected = '---y-';
const expectedMap = {
    x: 1,
    y: 2,
    z : 3
};

const lhs$ = testScheduler.createHotObservable(lhsMarble, { x: 1, y: 2, z :3 });

const myAlgorithm = ( lhs ) =>
    Rx.Observable
    .from( lhs )
    .filter(x => x % 2 === 0 );

const actual$ = myAlgorithm( lhs$ );

//assert
testScheduler.expectObservable(actual$).toBe(expected, expectedMap);
testScheduler.flush();
```

In this case our algorithm consists of a `filter()` operation. Which means 1,2,3 will not be emitted, only 2. Looking at the ingoing pattern we have:

```
'-x-y-z'
```

And expected pattern

```
`---y-`
```

And this is where you clearly see that no of `-` matters. Every symbol you write, be it `-` or `x` etc, happens at a certain time, so in this case when `x` and `z` wont occur due to the `filter()` method it means we just replace them with `-` in the resulting output so

```
-x-y
```

becomes

```
---y
```

because `x` doesn't happen.

There are of course other symbols that are of interest that lets us define things like an error. An error is denoted as a `#` and below follows an example of such a test:

```
const lhsMarble = '-#';
const expected = '#';
const expectedMap = {
};

//const lhs$ = testScheduler.createHotObservable(lhsMarble, { x: 1, y: 2, z :3 });

const myAlgorithm = ( lhs ) =>
    Rx.Observable
    .from( lhs );

const actual$ = myAlgorithm( Rx.Observable.throw('error') );

//assert
testScheduler.expectObservable(actual$).toBe(expected, expectedMap);
testScheduler.flush();
```

And here is another symbol `|` representing a stream that completes:

```
const lhsMarble = '-a-b-c-|';
const expected = '-a-b-c-|';
const expectedMap = {
    a : 1,
    b : 2,
    c : 3
};

const myAlgorithm = ( lhs ) =>
    Rx.Observable
    .from( lhs );

const lhs$ = testScheduler.createHotObservable(lhsMarble, { a: 1, b: 2, c :3 });
const actual$ = lhs$;

testScheduler.expectObservable(actual$).toBe(expected, expectedMap);
testScheduler.flush();
```

and there are more symbols than that like `(ab)` essentially saying that these two values are emitted on the same time frame and so on. Now that you hopefully understand the basics of how symbols work I urge you to write your own tests to fully grasp it and learn the other

symbols presented at the official docs page that I mentioned in the beginning of this chapter.

Happy testing

Cascading calls

A cascading call means that based on what call is happening another call should take place and possibly another one based on that.

Dependant calls

A dependant call means the calls needs to happen in order. Call 2 must happen after Call 1 has returned. It might even be possible that Call2 needs be specified using data from Call 1.

Imagine you have the following scenario:

- A user needs to login first
- Then we can fetch their user details
- Then we can fetch their orders

Promise approach

```
login()
  .then(getUserDetails)
  .then(getOrdersByUser)
```

Rxjs approach

```
import { of, from } from 'rxjs';
import { switchMap } from 'rxjs/operators';

let stream$ = of({ message : 'Logged in' }).pipe(
  switchMap( result => {
    return of({ id: 1, name : 'user' })
  }),
  switchMap((user) => {
    return from(
     [{ id: 114, userId : 1 },
      { id: 117, userId : 1 }  ])
    })
);

stream$.subscribe((orders) => {
  console.log('Orders', orders);
})

// Array of orders
```

I've simplied this one a bit in the Rxjs example but imagine instead of

```
of()
```

it does the proper `ajax()` call like in Operators and Ajax

Semi dependant

- We can fetch a users details
- Then we can fetch Orders and Messages in parallell.

Promise approach

```
getUser()
  .then((user) => {
    return Promise.all(
      getOrders(),
      getMessages()
    )
   })
```

Rxjs approach

```
import { of } from 'rxjs';
import { switchMap, forkJoin } from 'rxjs/operators';

let stream$ = of({ id : 1, name : 'User' })
stream.pipe(
  switchMap((user) => {
    return forkJoin(
      from([{ id : 114, user: 1}, { id : 115, user: 1}],
      from([{ id : 200, user: 1}, { id : 201, user: 1}])
    )
  })
);

stream$.subscribe((result) => {
  console.log('Orders', result[0]);
  console.log('Messages', result[1]);
})
```

GOTCHAS

We are doing `switchMap()` instead of `flatMap()` so we can abandon an ajax call if necessary, this will make more sense in Auto complete recipe

Excellent code example by the legendary @mattpodwysocki

https://github.com/Reactive-Extensions/RxJS/blob/master/examples/autocomplete/autocomplete.js

Great article on infinite scroll

https://hackernoon.com/naive-infinite-scroll-in-reactive-programming-using-rxjs-observables-4a605d3146e8

Another great article on the same topic

http://blog.brecht.io/infinite-scroll-with-rxjs-and-angular2/

Subject

A Subject is a double nature. It has both the behaviour from an Observer and an Observable. Thus the following is possible:

Emitting values

```
subject.next( 1 )
subject.next( 2 )
```

Subscribing to values

```
const subscription = subject.subscribe( (value) => console.log(value) )
```

To sum it up the following operations exist on it:

```
next([value])
error([error message])
complete()
subscribe()
unsubscribe()
```

Acting as a proxy

A `Subject` can act as a proxy, i.e receive values from another stream that the subscriber of the `Subject` can listen to.

```
import { interval, Subject } from 'rxjs';
import { take } from 'rxjs/operators';

let source$ = interval(500).pipe(
  take(3)
 );

const proxySubject = new Subject();
let subscriber = source$.subscribe( proxySubject );

proxySubject.subscribe( (value) => console.log('proxy subscriber', value ) );

proxySubject.next( 3 );
```

So essentially proxySubject `listens` to `source$`

But it can also add its own contribution

```
proxySubject.next( 3 )  // emits 3 and then 0 1 2 ( async )
```

GOTCHA
Any `next()` that happens before a subscription is created is lost. There are other Subject types that can cater to this below.

Business case

So what's interesting about this? It can listen to some source when that data arrives as well as it has the ability to emit its own data and all arrives to the same subscriber. Ability to communicate between components in a bus like manner is the most obvious use case I can think of. Component 1 can place its value through `next()` and Component 2 can subscribe and conversely Component 2 can emit values in turn that Component 1 can subscribe to.

```
sharedService.getDispatcher = () => {
   return subject;
}

sharedService.dispatch = (value) => {
  subject.next(value)
}
```

ReplaySubject

prototype:

```
import { ReplaySubject } from 'rxjs';

new ReplaySubject([bufferSize], [windowSize], [scheduler])
```

example:

```
import { ReplaySubject } from 'rxjs';

let replaySubject = new ReplaySubject( 2 );

replaySubject.next( 0 );
replaySubject.next( 1 );
replaySubject.next( 2 );

//  1, 2
let replaySubscription = replaySubject.subscribe((value) => {
    console.log('replay subscription', value);
});
```

Wow, what happened here, what happened to the first number?
So a `.next()` that happens before the subscription is created, is normally lost. But in the case of a `ReplaySubject` we have a chance to save emitted values in the cache. Upon creation the cache has been decided to save two values.

Let's illustrate how this works:

```
replaySubject.next( 3 )
let secondSubscriber( (value) => console.log(value) ) // 2,3
```

GOTCHA
It matters both when the `.next()` operation happens, the size of the cache as well as when your subscription is created.

In the example above it was demonstrated how to use the constructor using `bufferSize` argument in the constructor. However there also exist a `windowSize` argument where you can specify how long the values should remain in the cache. Set it to `null` and it remains in the cache indefinite.

Business case

It's quite easy to imagine the business case here. You fetch some data and want the app to remember what was fetched latest, and what you fetched might only be relevant for a certain time and when enough time has passed you clear the cache.

AsyncSubject

```
import { AsyncSubject } from 'rxjs';

let asyncSubject = new AsyncSubject();
asyncSubject.subscribe(
    (value) => console.log('async subject', value),
    (error) => console.error('async error', error),
    () => console.log('async completed')
);

asyncSubject.next( 1 );
asyncSubject.next( 2 );
```

Looking at this we expect 1,2 to be emitted right? WRONG.
Nothing will be emitted unless `complete()` happen

```
asyncSubject.next( 3 )
asyncSubject.complete()

// emit 3
```

`complete()` needs to happen regardless of the finishing operation before it succeeds or fails so

```
asyncSubject( 3 )
asyncSubject.error('err')
asyncSubject.complete()

// will emit 'err' as the last action
```

Business case

When you care about preserving the last state just before the stream ends, be it a value or an error. So NOT last emitted state generally but last *before closing time*. With state I mean value or error.

BehaviourSubject

This Subject emits the following:

- the initial value
- the values emitted generally
- last emitted value.

methods:

```
next()
complete()
constructor([start value])
getValue()
```

```
import { BehaviorSubject } from 'rxjs';

let behaviorSubject = new BehaviorSubject(42);

behaviorSubject.subscribe((value) => console.log('behaviour subject',value) );
console.log('Behaviour current value',behaviorSubject.getValue());
behaviorSubject.next(1);
console.log('Behaviour current value',behaviorSubject.getValue());
behaviorSubject.next(2);
console.log('Behaviour current value',behaviorSubject.getValue());
behaviorSubject.next(3);
console.log('Behaviour current value',behaviorSubject.getValue());

// emits 42
// current value 42
// emits 1
// current value 1
// emits 2
// current value 2
// emits 3
// current value 3
```

Business case

This is quite similar to `ReplaySubject` . There is a difference though, we can utilize a default / start value that we can show initially if it takes some time before the first values starts to arrive. We can inspect the latest emitted value and of course listen to everything that has been emitted. So think of `ReplaySubject` as more *long term memory* and `BehaviourSubject` as short term memory with default behaviour.

Schedulers

Tools

The idea with this section is to list nice tools that helps you when writing RxJS code

RxJS DevTools

Found on github link https://github.com/kwintenp/rx-devtools The README lists how to install the the npm/ yarn module and also the chrome plugin.

Very nice tool for visualising what your code does and what values will be emitted.

RxFiddle

Just go to the page http://rxfiddle.net/#type=editor and start writing your RxJS expressions. It will show you a visual of what a run looks like. It doesn't get any simpler.

ngrx

Build your own RxJS

To truly understand Rxjs I recommend trying to implement a subset of it yourself just to see what kind of code you produce. For most cases it isn't really necessary to understand on that depth but do try it if you got some time over. Let's look at what a really basic implementation can look like and refactor it slowly into something that looks like the real thing.

Step I

Step II

Step III

Step IV